Lexical Analysis

By:
Trusha R. Patel
Asst. Prof.
CE Dept, CSPIT, CHARUSAT

Role of Lexical Analyzer

1. Main task is to read input characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the program

2. Interact with symbol table, when it discovers a lexeme constituting an identifier, it need to enter that lexeme into the symbol table

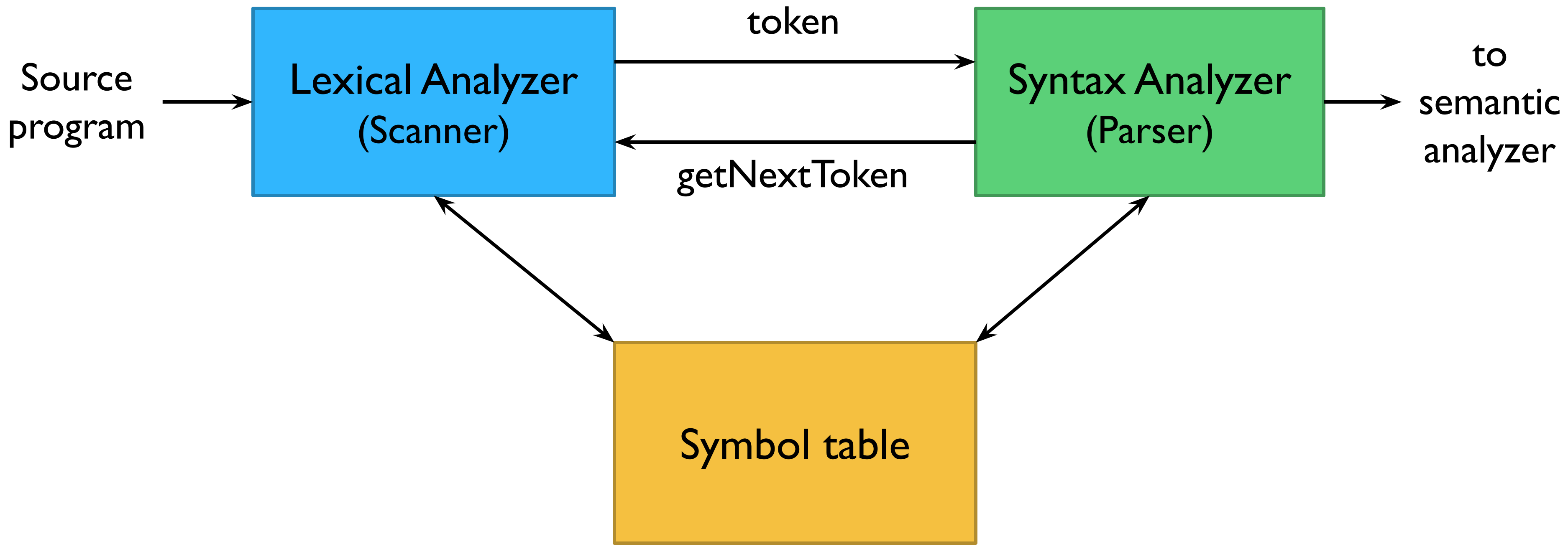
Role of Lexical Analyzer

3. Stripping out comments and whitespace

4. Generate error messages (lexical errors)

5. Keep track of the line number so it can associate a line number with each error

Interaction of Lexical Analyzer from Syntax Analyzer



Separated Lexical Analyzer from Syntax Analyzer

1. Simplicity of design
   e.g. its complex for parser to deal with comments and whitespaces as syntactic unit so they are removed in lexical analysis

2. Compiler efficiency is improved as it allows to apply specialized techniques that serve only the lexical task not parsing job
   specialized input buffering techniques for reading input characters can speed up the compiler

Separated Lexical Analyzer from Syntax Analyzer

3. Compiler portability is enhanced
   input-device-specific peculiarities can be restricted to the lexical analysis

Token, Pattern, Lexeme

- Token
  - ❖ It's a pair consisting of token name and attribute value
  - ❖ Generally write token name in boldface
  - ❖ Refer to a token by its name

- Pattern
  - ❖ Description of the form that lexemes of a token may take

- Lexeme
  - ❖ Sequence of characters in the source program that matches the pattern for a token

Token, Pattern, Lexeme

Token	Informal description	Sample lexeme
if	Characters i , f	if
else	Characters e , l , s , e	else
comparison	< or > or <= or >= or == or !=	<= , =>
id	Letter followed by letters and digits	pi , score , D2
number	Any numeric constant	3.14159 , 6.02e23
literal	Anything but " surrounded by "	"core dumped"

Attributes for Token

- When more then one lexemes can match a pattern, lexical analyzer must provide additional information to the subsequent compiler phases
- So lexical analyzer return token name with attribute value
- Token have at most one associated attribute, although this attribute may have a structure that combined several information

Attributes for Token

- Example
  - Token **id**
  - Information about identifier is kept in symbol table
  - Attribute value for identifier is a pointer to the symbol table entry for that identifier

Attributes for Token

- Example
  - ❖ Name and associated attribute value for E = M * C ** 2

    < **id** , pointer to symbol-table entry of E >
    < **assign-op** >
    < **id** , pointer to symbol-table entry of M >
    < **mult-op** >
    < **id** , pointer to symbol-table entry of C >
    <**exp-op**>
    < **number** , integer value 2 >

Lexical Error

- **fi** is encountered for the first time in C :
  fi (a== f(x)) …
  then lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or undefined function identifier

- Since **fi** is valid lexeme for token id the lexical analyzer return token id to the parser and parser handle the error

Lexical Error

- If lexical analyzer is unable to proceed because none of the patterns of token matches any prefix of the remaining input

- Simplest recovery strategy is "panic mode" recovery delete successive characters from the remaining input until the lexical analyzer can find a well-formed token

Lexical Error

- Other possible error-recovery actions are
  - ❖ Delete one character from the remaining input
  - ❖ Insert a missing character into the remaining input
  - ❖ Replace a character by another character
  - ❖ Transpose two adjacent characters

Input Buffering

- In C, single-character operator like > , = , < can also be the beginning of two-character operator >= , == , <=

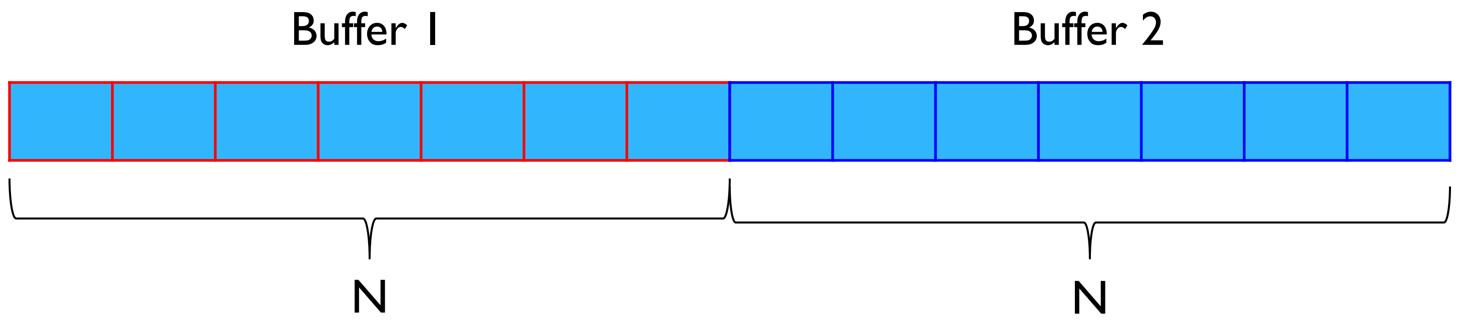
- Often have to look one or more characters beyond the next lexeme before we can be sure about correct lexeme

- Introduce a two-buffer scheme that handle large lookaheads

Buffer Pairs

- Large amount of time taken to process characters
  This buffering technique have been developed to reduce the amount of overhead required to process a single input character

- It involves two buffer that are alternately reloaded
  Each buffer is of the same size N (usually the size of disk block)

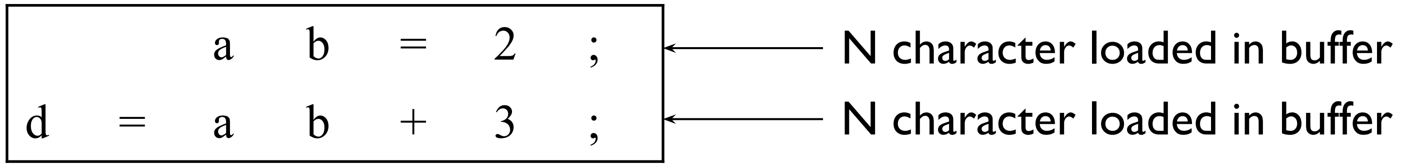


Buffer Pairs

- One system read command will read N characters into buffer instead of one character
  If fewer than N characters remain in input file,
  then special character **eof** marks the end of source file

- Maintains two pointer
  - lexemeBegin
    - Marks beginning of current lexeme
  - forward
    - Scan ahead until a pattern match is found

Buffer Pairs



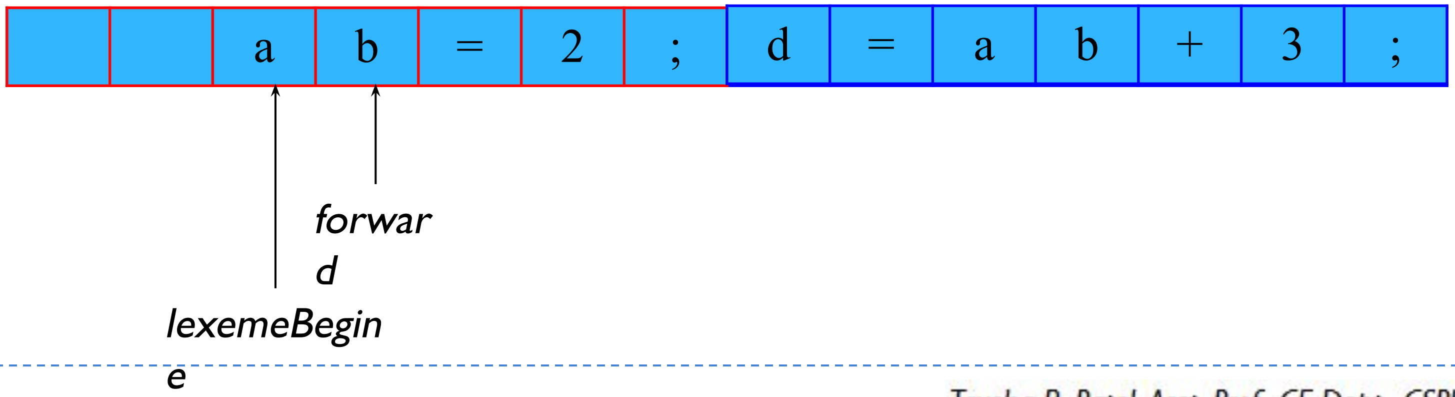
Match with **id**
Start of new token, set forward to one left position

id token generated for **ab** , now set lexemeBegine to next position of forward

Same way tokens generated for **=** , **2** , **;**

then pointer reach to end of first buffer so next buffer will be loaded

Set points in new buffer and do the same process for the next buffer



Symbol, Alphabet, String, Language

- Symbol
  - ❖ It can be letters, digits, punctuations, operators
  - ❖ E.g.
    a-z A-Z a-z 0-9 ; = + , etc.

- Alphabet (Σ)
  - ❖ Finite set of symbols
  - ❖ E.g.
    {0,1} { a , b , c } etc.

Symbol, Alphabet, String, Language

- String
  - ❖ String over alphabet is finite sequence of symbols drawn from that alphabet
  - ❖ E.g.
    for alphabet {0,1} 0101 can be a string
  - ❖ Empty string is denoted using ϵ

- Language (L)
  - ❖ Any countable set of strings over some fixed alphabet
  - ❖ E.g.
    for alphabet {0,1} language is set of string of length 2 can be given as { 00 , 01 , 10 , 11 }

Terms for Parts of String

- Prefix
  - String obtain by removing zero or more symbols from end of string
  - E.g. all possible prefixes of banana are:
    ϵ b ba ban bana banan banana
- Suffix
  - String obtain by removing zero or more symbols from beginning of string
  - E.g. all possible suffixes of banana are:
    banana anana nana ana na a ϵ

Terms for Parts of String

- Substring
  - ❖ Obtained by deleting any prefix and any suffix from string
  - ❖ E.g. some substrings of banana can be
    nan anan …
- Proper prefixes, suffixes and substrings
  - ❖ Prefix, suffix and substring are called proper if they are not ϵ or not equal to string itself

Terms for Parts of String

- Subsequence
  - ❖ Deleting zero or more not necessarily consecutive positions from string
  - ❖ E.g. substring of banana can be
    baan bn …

Operations on Languages

OPERATION	DEFINITION & NOTATION
Union of L and M	LUM = { s | s is in L or s is in M }
Concatenation of L and M	LM = { st | s is in L and t is in M }
Kleene closer of L	
Positive closer of L	

Here L and M are two languages

Regular Expression

Notation	Meaning
ϵ	Null character
|	or / Choice
*	Kleene closure 0 or more occurrence
+	Positive closure 1 or more occurrence
?	0 or 1 occurrence
( )	Concatenation

Notation	Meaning
{ }	Used to define rang of occurrence
ø	Null set
U	Union
[ ]	Character set
^	Reverse of set

Algebraic laws for Regular Expression

LAW	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
r (s t) = (r s) t	Concatenation is associative
r (s | t) = r s | r t ; (s | t) r = s r | t r	Concatenation distributes over |
ϵ r = r ϵ = r	ϵ is the identity for concatenation
r* = (r | ϵ)*	ϵ is guaranteed in a closure
r** = r*	* Is idempotent

Regular expression

- RE for identifier

  [a-z A-Z _] [a-z A-Z _ 0-9] *

Regular Definition

- Allow to give names to certain regular expressions and that name can be used in subsequent expression
- Regular definition is sequence of definitions of the form

  d1 → r1

  d2 → r2

  …

  dn → rn

  where

  1) each di is new symbol, not in Σ and not same as any d's

  2) each ri is a regular expression over
     alphabet Σ ∪ { d1 , d2 , … , dn }

Regular Definition

- Regular definition for identifier

  letter □ A | B | … | Z | a | b | … | z | _
  digit □ 0 | 1 | 2 | … | 9
  id □ letter (letter | digit)*

- Using shorthands

  letter □ [A-Z a-z _]
  digit □ [0 – 9]
  id □ letter (letter | digit)*

Regular Definition

- Regular definition for unsigned number

digit	□	0 | 1 | … | 9
digits	□	digit digit *
optionalFraction	□	. digits | ϵ
optionalExponent	□	(E (+ | - | ϵ) digits) | ϵ
number	□	digits optionalFraction optionalExponent

- Using shorthands

digit	□	[0 – 9]
digits	□	digit^{+}
number	□	digits (. digits) ? (E [+ -] ? digits)?

Transition Diagram

- Intermediate state in construction of lexical analyzer is conversion of patterns into stylish flowchart called "transition diagram"

- Transition diagram have a collection of nodes or circles called "states"
  Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several pattern

Transition Diagram

- "Edges" are directed from one state to other, labeled by a symbol or set of symbols
  Is current state is "s" and input symbol is "a" then find edge out from "s" with label "a", if found then advance the forward pointer in buffer and enter the state of transition diagram which that edge leads
  Assume that all transition diagrams are deterministic means there is never more than one edge out of a state with a given symbol among its labels

Transition Diagram

- “Start state” or “initial state” indicated by edge labeled “start”
  Transition diagram always begin in the start state

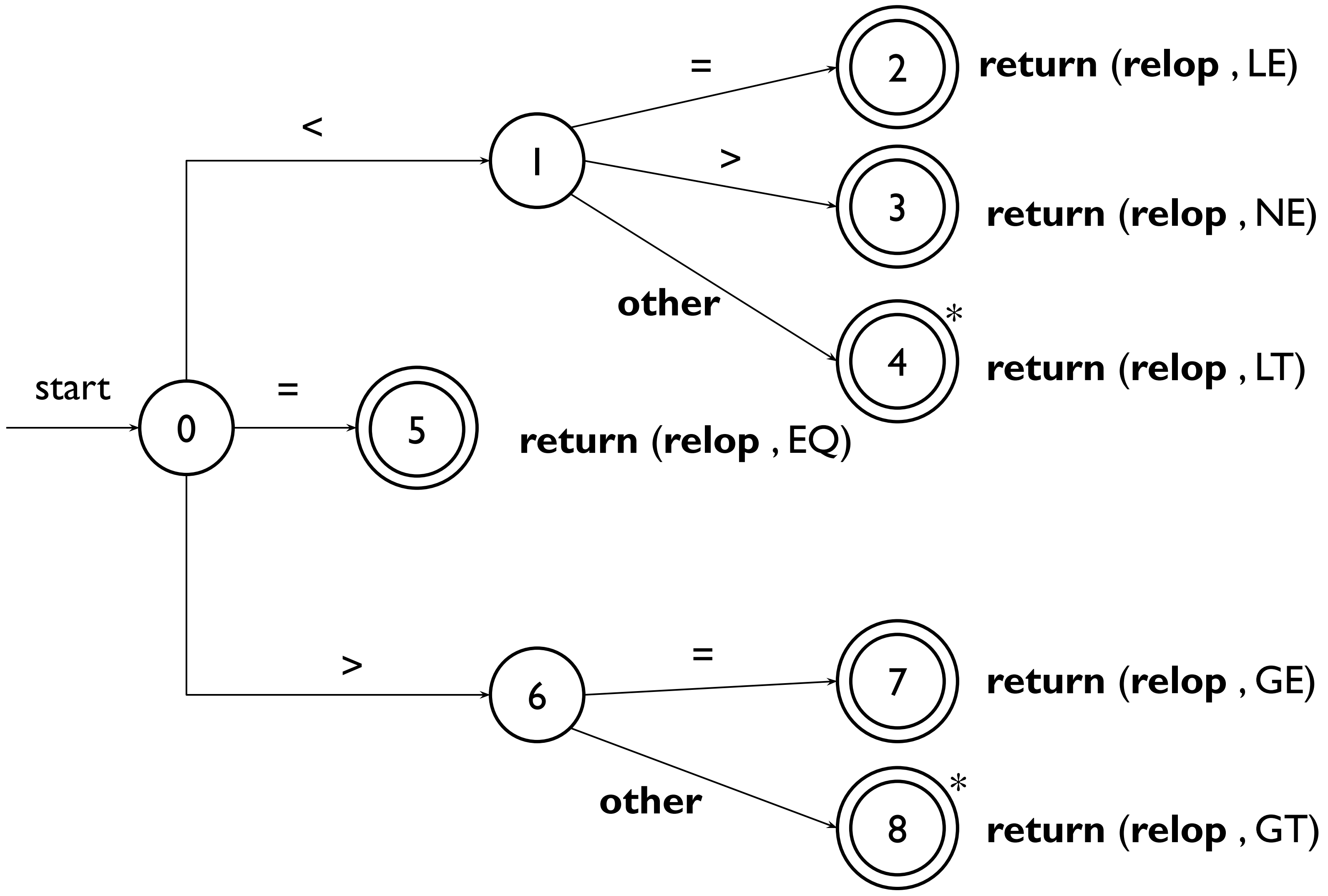
- Certain states are called “accepting state” of “final state”, indicated the lexeme has been found and indicated by double circle, if any action to be taken need to attach that action o accepting state

- If it is necessary to retract the forward pointer (at that point of time you can not decide that token can be generated or not so you need to check one or more characters more) then additionally place * near accepting state

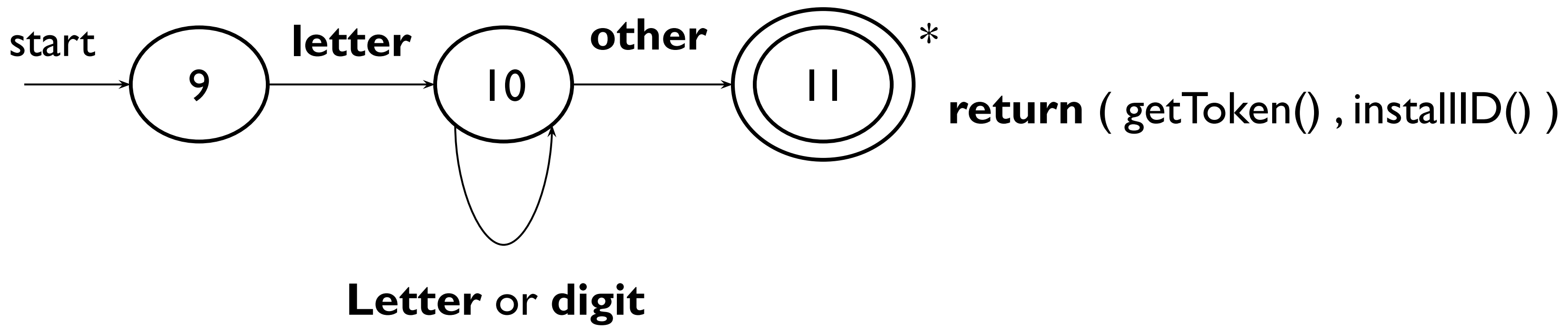
Transition Diagram

LEXEME	TOKEN NAME	ATTRIBUTE VALUE
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE
if	if	-
then	then	-
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
Any ws	-	-

Transition Diagram (Recognition relational operation)



Transition Diagram (Recognition identifier and keyword)



- Problem is to differentiate keyword and identifier

Transition Diagram (Recognition identifier and keyword)

- Two solutions

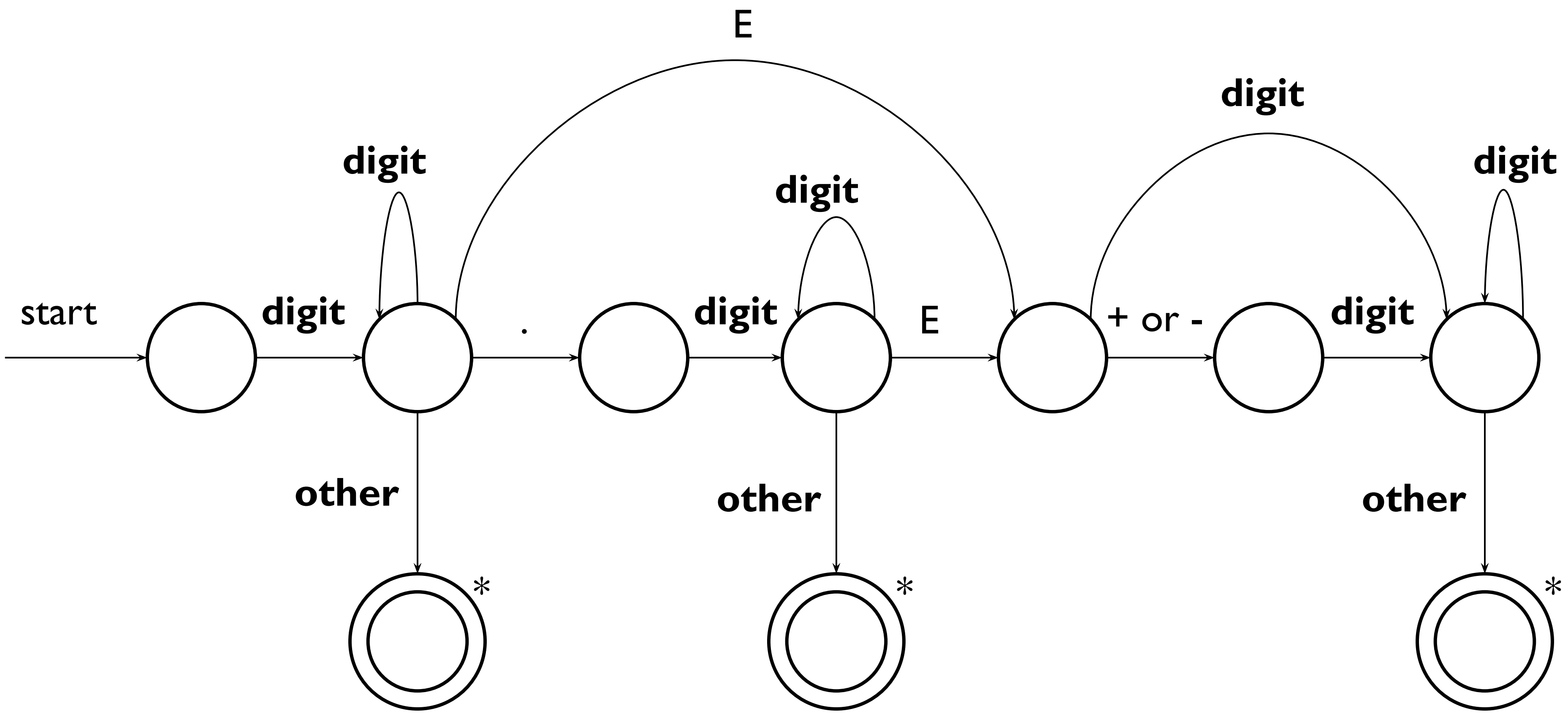
❖ Install keyword in symbol-table initially

- One field of symbol-table indicate that it's a keyword or identifier
  when pattern match then getToken find in symbol-table
  if found then return pointer to its symbol –table entry
  if not then installID enter that identifier in symbol-table and return the pointer to that entry

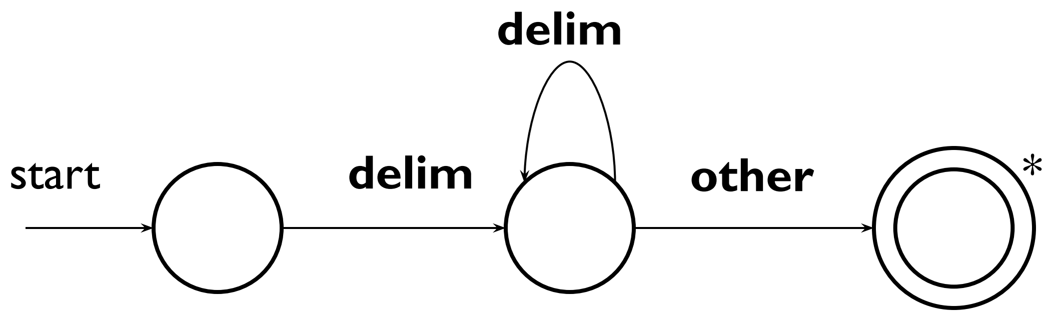
❖ Create separate transition diagram for each keyword

- all edge show successive letters of keyword
  last test for "nonletter-or-digit"

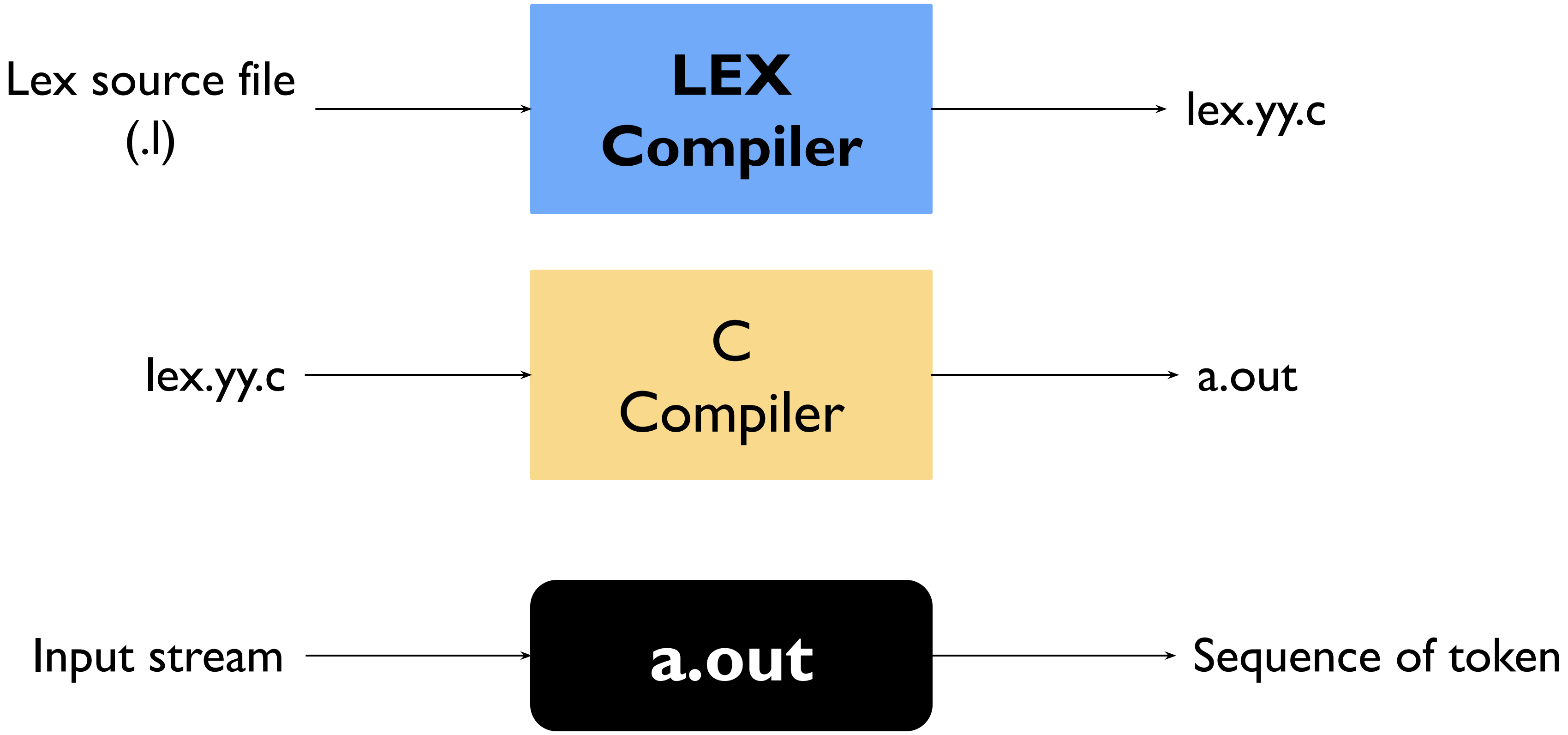
Transition Diagram (Recognition unsigned number)



Transition Diagram (Recognition whitespace)



Lexical Analyzer Generator LEX



Structure of Lex Program

declarations

%%

translation rules

%%

Auxiliary functions

Declaration of variables, constants and regular definitions

of form :- Pattern { Action }
where
Pattern :- regular expression which may use regular definition from declaration section
Action :- fragment of code written in C

Hold additional functions used in the actions

Finite Automata

- They are recognizers
- Simply say "yes" or "no" about each possible input string
- Two flowers
  - ❖ NFA (Nondeterministic Finite Automata)
    - A symbol can label several edges out of he same state, ϵ is also a possible label
  - ❖ DFA (Deterministic Finite Automata)
    - For each symbol exactly one edge with that symbol leaving that state

NFA

- Consists of

S	finite set of states
Σ	set of input symbols ,called input alphabet
Transition function	gives for each state and each input symbol a set of next states
s_0	Start state or initial state , from S
F	Set of accepting states , subset of S

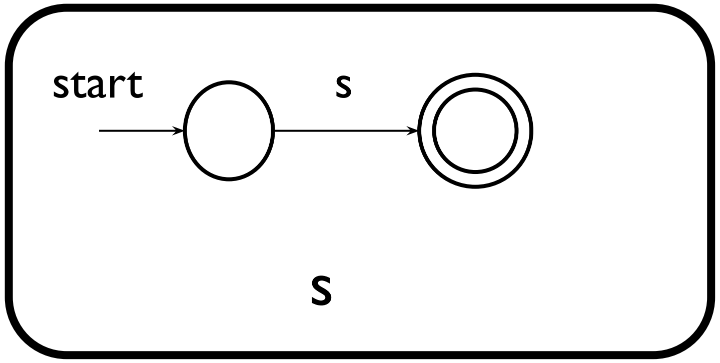
DFA

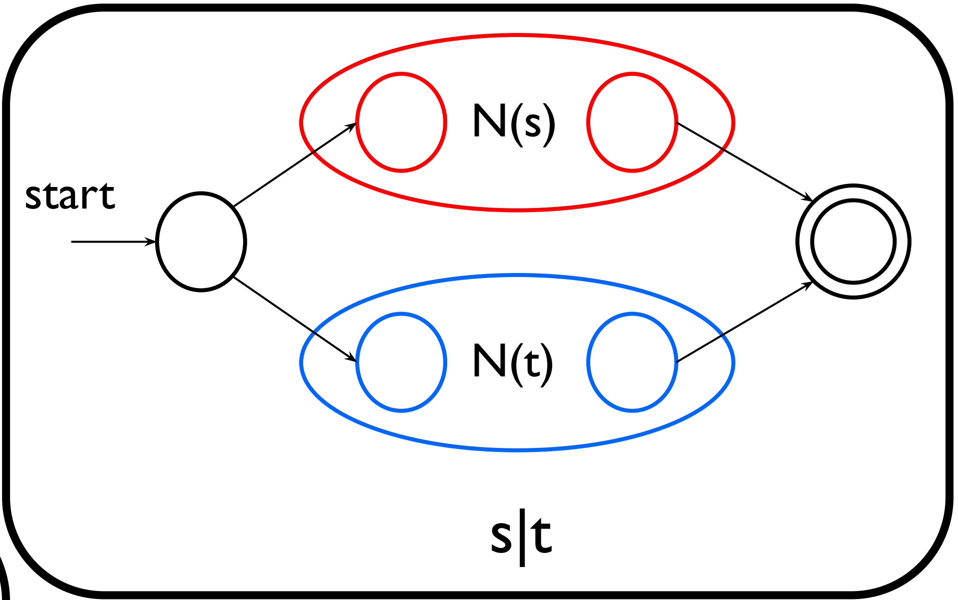
- Special case of NFA
  - ❖ No moves on input ϵ
  - ❖ Each state “s” and input symbol “a”, exactly one edge out of “s” labeled with “a”

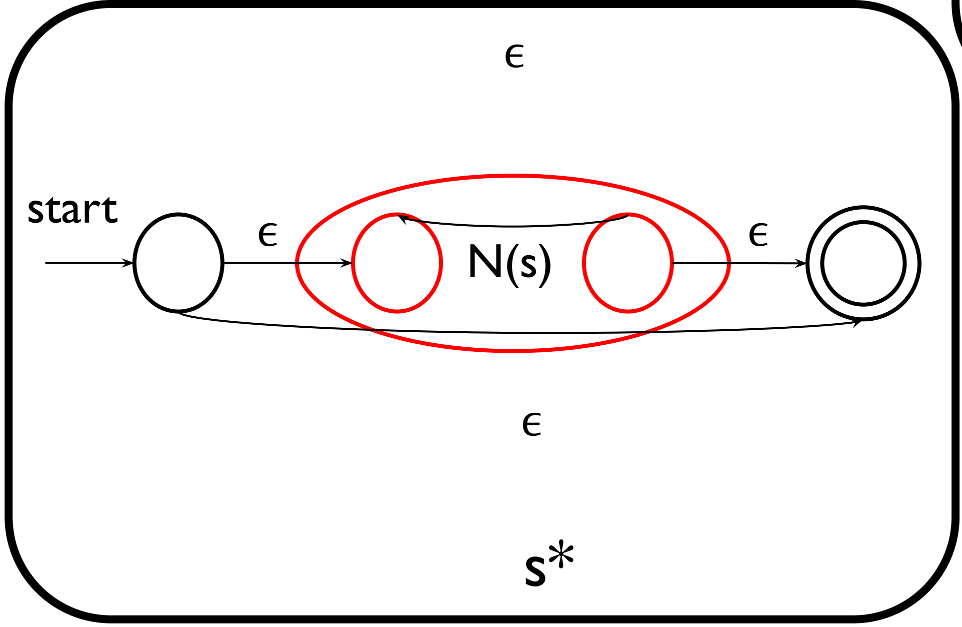
RE to NFA (using Thompson Construction)

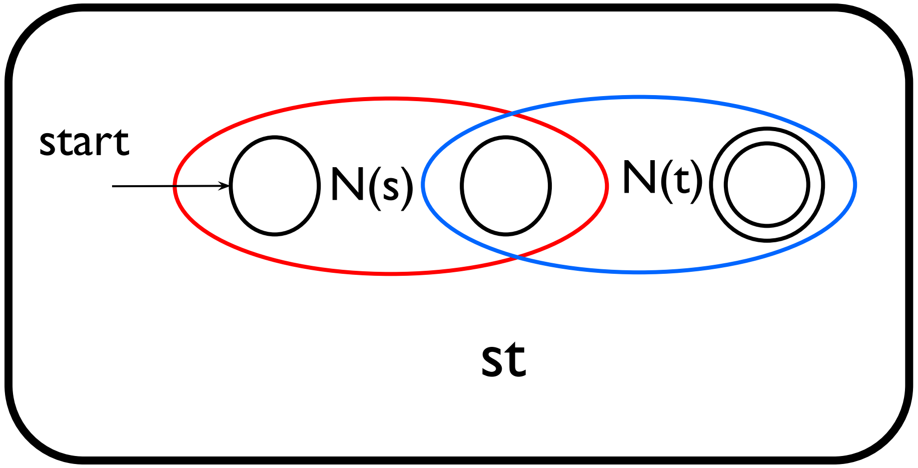
- McNaughton-Yamada-Thompson algorithm
  - ❖ Works recursively
  - ❖ For each subexpression it constructs an NFA with a single accepting state

RE to NFA (using Thompson Construction)



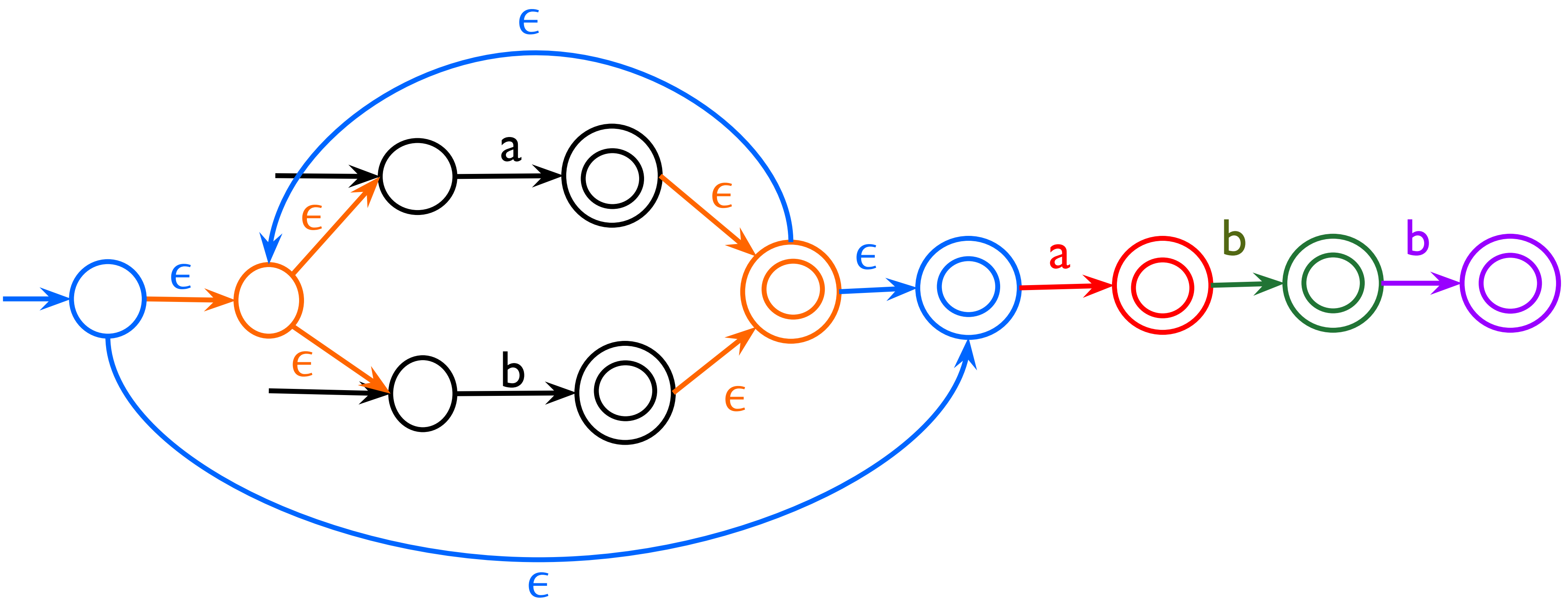






RE to NFA (using Thompson Construction)

(a | b) * a b b



RE to NFA

- More examples
  - ❖ $(a \mid b)^*$
  - ❖ $a^* \mid b^*$
  - ❖ $(a^* \mid b)^* c a^*$
  - ❖ $(a \mid b)^* c^* d$
  - ❖ $a^* b a$
  - ❖ $a^+ b^* (c \mid d)$
  - ❖ $a^+ b (c \mid d) a^* b$
  - ❖ $((a^* b^*) c)^* a^*$
  - ❖ $(a \mid b)^* a b b$
  - ❖ $a a^* \mid b b^*$

RE to DFA

- Steps

1. Add # at the end of RE
2. Construct tree of RE
3. Give numbers (starting with 1) to all alphabet
4. Find FIRSTPOS for all position in constructed tree
5. Find FOLLOWPOS for all position in constructed tree
6. Take highest FIRSTPOS set as A state
7. Find Next state for all possible input symbol from A
8. If new set comes then give it new name (i.e. B, C, D, …)
9. Continue till new won't get any new set
10. Make transition table
11. Construct DFA from transition table
12. Remove transition with #
13. Change final state accordingly
14. If from any state transitions are missing for any symbol then create dead end ant connect missing transitions with it

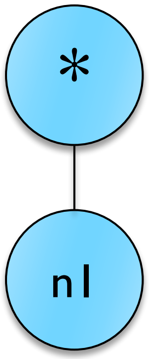
nullable( ) , fisrtpos( ) , followpos( )

- nullable(n)
  - ❖ If "n" is a leaf node labeled ϵ then nullable(n) is TRUE
    If "n" is a leaf node labeled "n" then nullable(n) is FALSE
- firstpos(n)
  - ❖ Set of symbols or positions that can come as first of subexpression appearing at position "n"
- followpos(n)
  - ❖ Set of symbols or positions that can follow the subexpression appearing at position "n"

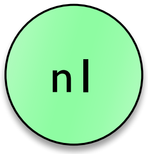
nullable( ) , fisrtpos( ) , followpos( )

NODE n	nullable(n)	firstpos(n)
A leaf labeled ϵ	true	Ø
A leaf with position i	false	{ i }
An or-node n=c1|c2	nullable(c1) or nullable(c2)	firstpos(c1)Ufirstpos(c2)
A cat-node n=c1c2	nullable(c1) and nullable(c2)	if (nullable(c1)) fisrtpos(c1) Ufirstpos(c2) else firstpos(c1)
A star-node n=c1*	true	firstpos(c1)

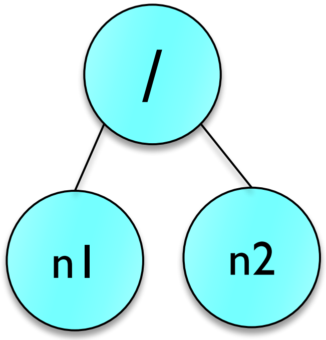
nullable( ) , fisrtpos( ) , followpos( )



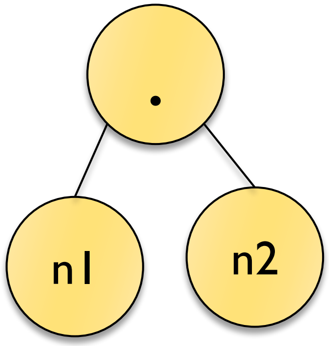
FIRSTPOS (*) = FIRSTPOS (n1)



FIRSTPOS (n1) = n1



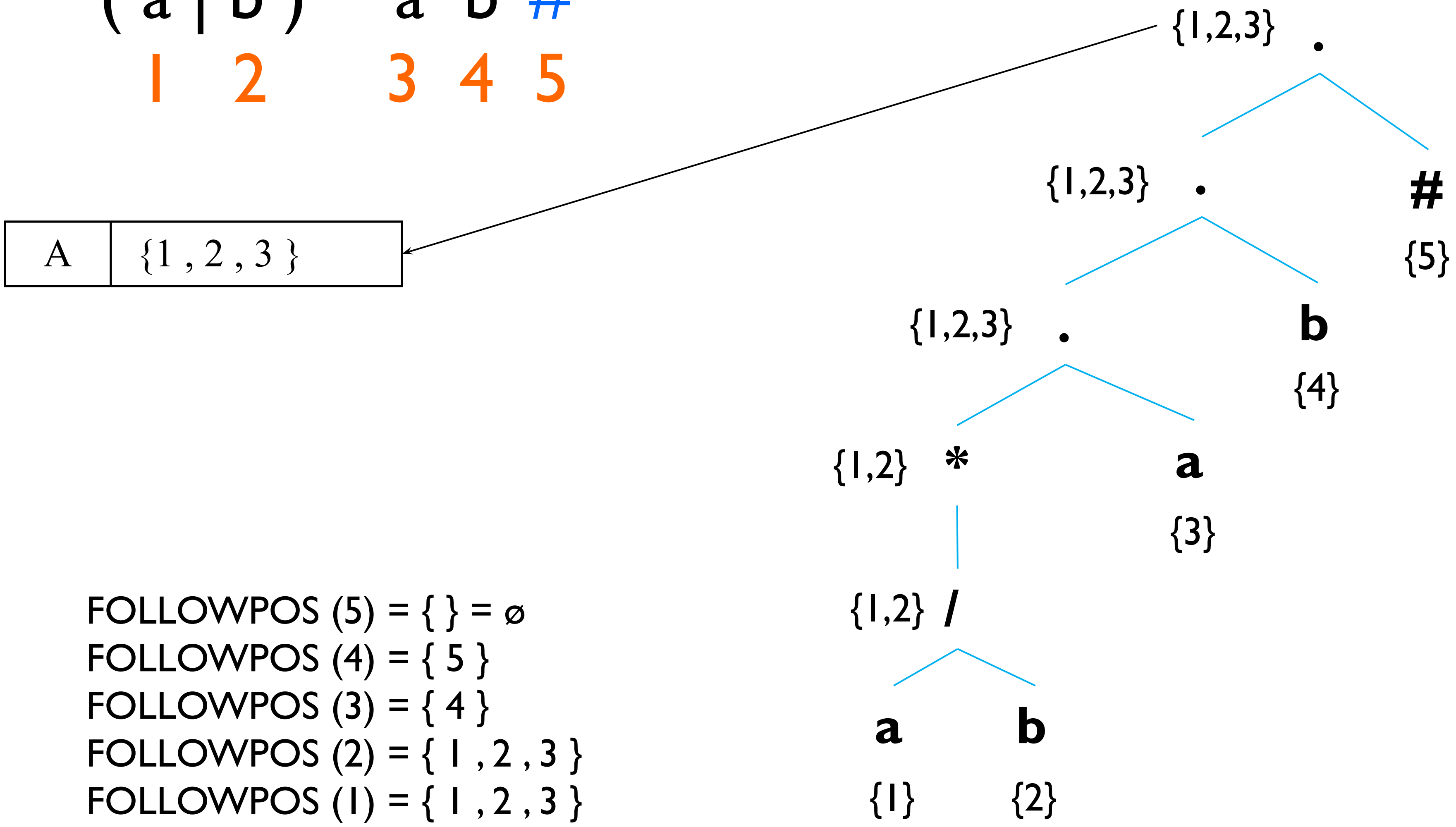
FIRSTPOS (/) = FIRSTPOS (n1) U FIRSTPOS (n2)



FIRSTPOS (.) = FIRSTPOS (n1) U FIRSTPOS (n2) if n1is nullable
FIRSTPOS (.) = FIRSTPOS (n1) if n1 is not nullable

RE to DFA (using fistpos, followpos, lastpos)

(a | b) * a b #
1 2 3 4 5



FOLLOWPOS (5) = { } = ⌀
FOLLOWPOS (4) = { 5 }
FOLLOWPOS (3) = { 4 }
FOLLOWPOS (2) = { 1 , 2 , 3 }
FOLLOWPOS (1) = { 1 , 2 , 3 }

RE to DFA (using fistpos, followpos, lastpos)

$$\underset{}{(}\underset{1}{a} \mid \underset{2}{b})^* \ \underset{3}{a}\ \underset{4}{b}\ \underset{5}{\#}$$

		a	b	#
A	{1 , 2 , 3 }	B	A	-
B	{1 , 2 , 3 , 4 }			

A { $\overset{}{\underset{a}{1}}$, $\underset{b}{2}$, $\underset{a}{3}$ }

i/p a = FOLLOW (1) U FOLLOW (3) = **{ 1 , 2 , 3 , 4 } = B**
i/p b = FOLLOW (2) = { 1 , 2 , 3 } = A
i/p # = ---

FOLLOWPOS (5) = { } = ø
FOLLOWPOS (4) = { 5 }
FOLLOWPOS (3) = { 4 }
FOLLOWPOS (2) = { 1 , 2 , 3 }
FOLLOWPOS (1) = { 1 , 2 , 3 }

RE to DFA (using fistpos, followpos, lastpos)

$$\begin{array}{ccccccc} (a & | & b) ^* & a & b & \# \\ 1 & & 2 & 3 & 4 & 5 \end{array}$$

		a	b	#
A	{1 , 2 , 3 }	B	A	-
B	{1 , 2 , 3 , 4 }	B	C	-
C	{1 , 2 , 3 , 5 }			

B { 1 , 2 , 3 , 4 }

a b a b

i/p a = FOLLOW (1) U FOLLOW (3) = { 1 , 2 , 3 , 4 } = B
i/p b = FOLLOW (2) U FOLLOW (4) = **{ 1 , 2 , 3 , 5 } = C**
i/p # = ---

FOLLOWPOS (5) = { } = ø
FOLLOWPOS (4) = { 5 }
FOLLOWPOS (3) = { 4 }
FOLLOWPOS (2) = { 1 , 2 , 3 }
FOLLOWPOS (1) = { 1 , 2 , 3 }

RE to DFA (using fistpos, followpos, lastpos)

(a | b) * a b #

1 2 3 4 5

		a	b	#
A	{1 , 2 , 3 }	B	A	-
B	{1 , 2 , 3 , 4 }	B	C	-
C	{1 , 2 , 3 , 5 }	B	A	D
D	{ }			

C { 1 , 2 , 3 , 5 }

a b a #

i/p a = FOLLOW (1) U FOLLOW (3) = { 1 , 2 , 3 , 4 } = B
i/p b = FOLLOW (2) = A
i/p # = FOLOW (5) = { } = D

FOLLOWPOS (5) = { } = ø
FOLLOWPOS (4) = { 5 }
FOLLOWPOS (3) = { 4 }
FOLLOWPOS (2) = { 1 , 2 , 3 }
FOLLOWPOS (1) = { 1 , 2 , 3 }

RE to DFA (using fistpos, followpos, lastpos)

(a | b) * a b #

1 2 3 4 5

		a	b	#
A	{1 , 2 , 3 }	B	A	-
B	{1 , 2 , 3 , 4 }	B	C	-
C	{1 , 2 , 3 , 5 }	B	A	D
D	{ }	-	-	-

D { }

i/p a = ---
i/p b = ---
i/p # = ---

FOLLOWPOS (5) = { } = ø
FOLLOWPOS (4) = { 5 }
FOLLOWPOS (3) = { 4 }
FOLLOWPOS (2) = { 1 , 2 , 3 }
FOLLOWPOS (1) = { 1 , 2 , 3 }

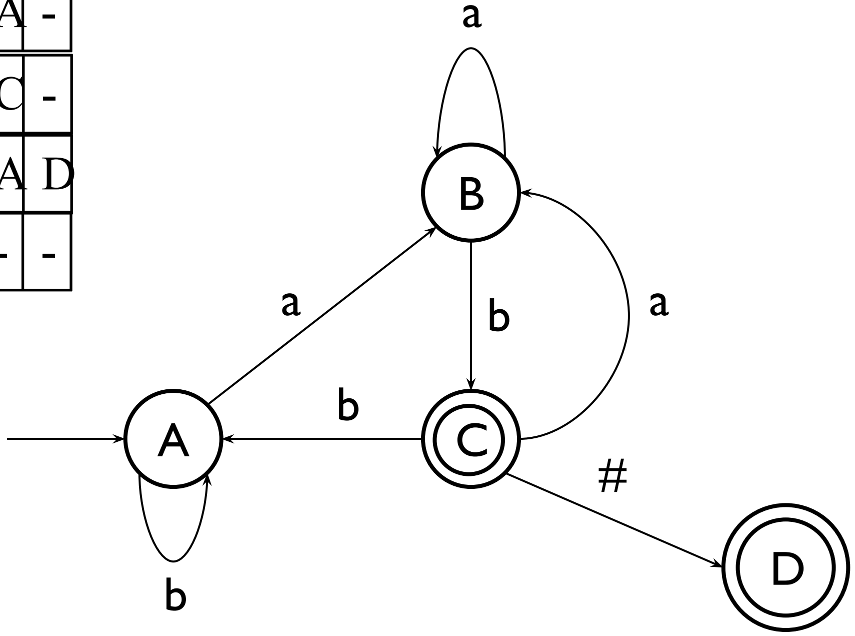
RE to DFA (using fistpos, followpos, lastpos)

$$\begin{array}{cccccccccc} (& a & | & b &)^{*} & a & b & \# \\ & 1 & & 2 & & 3 & 4 & 5 \end{array}$$

Initial state

		a	b	#
A	{1 , 2 , 3 }	B	A	-
B	{1 , 2 , 3 , 4 }	B	C	-
C	{1 , 2 , 3 , 5 }	B	A	D
D	{ }	-	-	-

Final state



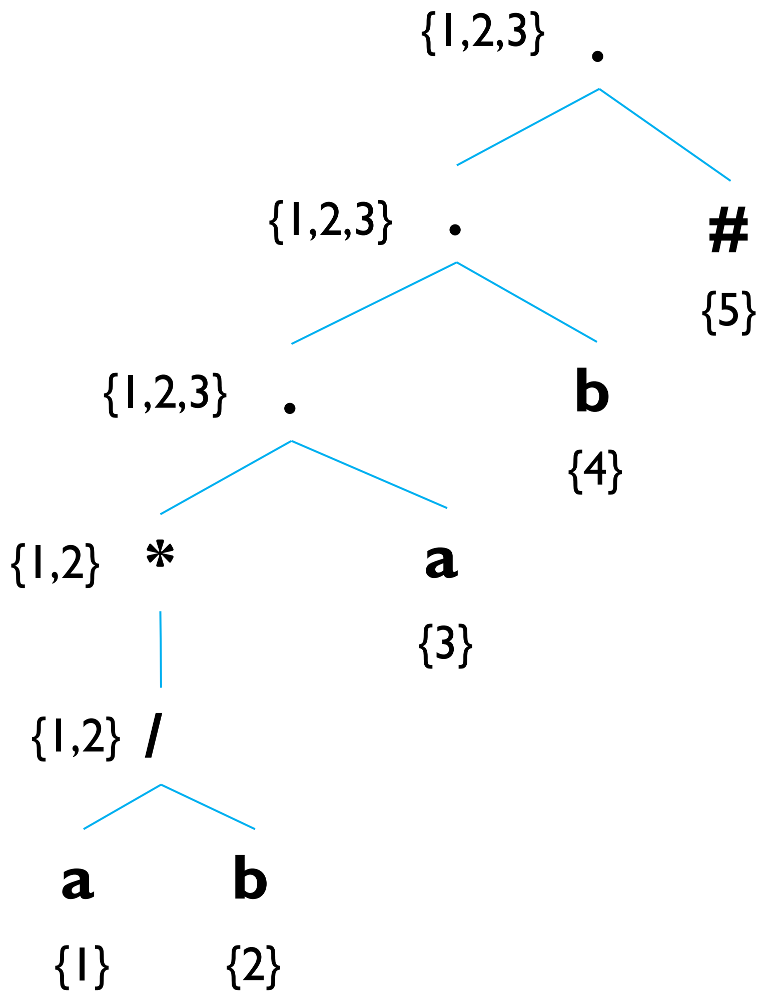
RE to DFA

- More examples
  - (a* | b*) c* a
  - ((a* b a) | (b* a)) (a b)*
  - (a | b)* a (a| b)
  - ((a* b*) c)* a*

RE to NFA (using fistpos, followpos, lastpos)

(a | b) * a b #

1 2 3 4 5



RE to NFA (using fistpos, followpos, lastpos)

(a | b) * a b #

1 2 3 4 5

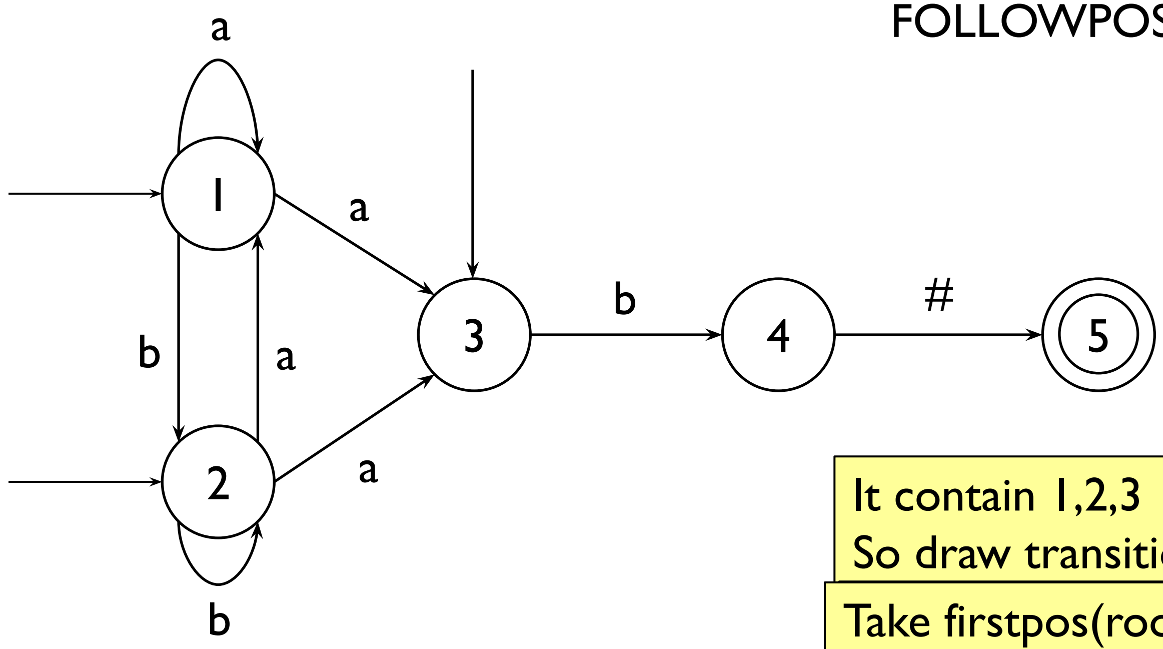
FOLLOWPOS (1) = { 1 , 2 , 3 }

FOLLOWPOS (2) = { 1 , 2 , 3 }

FOLLOWPOS (3) = { 4 }

FOLLOWPOS (4) = { 5 }

FOLLOWPOS (5) = { }



It contain 1,2,3

So draw transition from

Take firstpos(root)={1,2,3} as initial state

1 □2 (2 means "b" so label with "b")

1 □3 (3 means "a" so label with "a")

We can construct DFA from this NFA using subset construction method (covered in TOC)

END